INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

The Junior Reactive Kernel

Laurent Hazard — Jean-Ferdy Susini — Frédéric Boussinot

N° 3732

Juillet 1999

THÈME 1

ISSN 0249-6399

The Junior Reactive Kernel

Laurent Hazard* , Jean-Ferdy Susini , Frédéric Boussinot†

Thème 1 — Réseaux et systèmes
Projets Meije

Rapport de recherche n° 3732 — Juillet 1999 — 41 pages

Abstract: We define Junior as a Java framework kernel for reactive programming with broadcast events. We give Junior a formal semantics based on rewriting rule. We also describe three implementations of Junior. The first one, called REWRITE, is the direct implementation of the semantics rules. The second one is called REPLACE; it is more efficient than REWRITE as it reuses Java objects instead of always creating new ones. The third implementation, called TURBO, optimises the number of syntax tree traversals and is adapted to situations where there are a large number of events. Finally, we discuss the extension of Junior to distributed contexts and compare it with the SugarCubes Reactive Java framework.

Key-words: Reactive Systems, Semantics, Concurrency, SugarCubes, Java

Junior = **Jr**: **J** for Java and **r** for reactive... With support from France Telecom-CNET

* CNET DTR/ASR
† EMP-CMA/INRIA Meije

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles, B.P. 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : 04 92 38 77 77 - International : +33 4 92 38 77 77 —Fax : 04 92 38 77 65 - International : +33 4 92 38 77 65

Le noyau réactif Junior

Résumé : Nous définissons dans ce texte un noyau de primitives pour la programmation réactive en Java. Nous donnons la sémantique formelle de ce noyau à l'aide de règles de reécritures. Nous décrivons également trois implémentations de Junior. La première, appelée REWRITE, implémente directement les règles de la sémantique. La seconde, appelée REPLACE, est plus efficace que REWRITE car elle réutilise les objets au lieu d'en créer toujours de nouveaux. La troisième implémentation, appelée TURBO, optimise le nombre de parcours de l'arbre de syntaxe des programmes et est particulièrement efficace lorsqu'il y a un grand nombre d'événements. On termine en étendant Junior à la distribution et en le comparant aux SugarCubes.

Mots-clés : Systèmes réactifs, Sémantique, Parallélisme, SugarCubes, Java

1. Introduction

Reactive programming considers software systems which are:

- *Event-based.* In these systems, events are instantly broadcast. Communication is similar to radio transmissions, where emitters send information that is instantaneously received by all listeners. Using this communication paradigm, one can structure systems in a very modular way. For example, adding new receivers to a system is totally transparent and does not affect the others components (this is not the case with other communication mechanisms like message passing or *rendezvous*).

- *Parallel, but thread-less.* Parallelism is a logical programming construct to implement activities which are supposed to proceed concurrently and not one after the other. Such concurrent activities need not be executed by distinct threads, but instead can be automatically interleaved to get the desired result. This avoids some well-known problems related to threads.

- *Reactive.* Reactive systems are systems which continuously interact with their environment. A natural way to program these systems is to use reactive instructions with semantics defined by reference to activation/reaction pairs corresponding to instants. The end of the reaction initiated by a given activation naturally determines stable states, when a system is waiting exclusively for the next activation to resume execution.

The reactive approach defines *reactive instructions* and *reactive machines* for concurrent programming. A reactive instruction describes a *behaviour* and its associated *state*. The difference with threads is that a reactive instruction is intended to be executed by a reactive machine, not by a thread scheduler. More precisely:

- *Deterministic scheduling.* Reactive machines schedule reactive instructions in a *deterministic* way. Execution is totally independent from the platform; it does not depend on a particular scheduling strategy (cooperative or preemptive models). Execution is actually completely captured by reactive instructions semantics and does not depend on external characteristics such as priorities.

- *Complete control*. Reactive instructions and machines give programmers a sound and *fine control over execution*; for example, reactive instructions can be preempted in a totally clean way.

- *Automatic atomicity.* Reactive instructions provide automatic *atomic accesses* to data; there is no need of any synchronisation while accessing them.

Advantages over threads are the followings:

- *Full portability*. To write a truly platform-independent threading system is an extremely difficult task[Ho]. Reactive machines and reactive instructions they run map into *one unique* thread. This simplifies the portability task in a large extend.

- *Safe control*. Control over reactive instructions solves the problems raised in Java by `stop`, `suspend`, and `resume` methods of threads; see [JSD] for details on these problems.

- *Easier debugging.* Reactive instructions execution is predictable; this is a important good point for program debugging, as a faulty situation can be reproduced to be analysed.

- *Clear semantics*. Reactive instructions have a simple formal semantics, the existence of which opens the way to several optimisation and validation techniques (based on the notion of a finite states machine).

- *High-level communications.* Reactive instructions communicate using built-in *broadcast events* which provides the programmer with a safe, high-level, and powerful communication mechanism. This is to be compared to Java threads with which the programmer must implement communication protocols only using the low-level `wait`/`notify` synchronisation primitives.

Threads programming is actually a difficult exercise, in which one must deal with several overlapping notions, such as priorities, scheduling policies, or accesses to shared resources. Parallelism and the use of broadcast events in the reactive approach are conceptually much simpler. In this respect, Reactive programming can certainly be considered as a possible alternative to Java threads. This is particularly true when one needs dynamic combinations of communicating concurrent components.

SugarCubes[BS] is a proposal for reactive programming in Java. In this text, one considers a new model inspired by SugarCubes and called *Junior*. More precisely:

- one defines Junior as a small set of operators for reactive programming;

- one gives Junior formal semantics and describes several implementation of it;

- finally, one compares Junior with SugarCubes and some related formalisms.

2. The Junior Kernel

There are 3 basic notions in Junior: reactive instructions, events, and execution contexts. Reactive instructions basically react to activations. Several operators are available to combine reactive instructions reactions; one of them is the parallel operator which make several instructions react in response to a unique activation. Reactive instructions are run by execution contexts which defines their instants: instants of a reactive instruction actually consist in reactions to activations coming from the execution context running it. Events are used for reactive instruction communication and synchronisation; they are broadcast by execution contexts during instants.

2.1 Events

Reactive programming provides events with the following characteristics:

- events are automatically reset at the beginning of each new instant; thus, events are *not* persistent data spanning instants.
- an event is present during an instant if it is generated during this very instant. Generating an event which is already present has no effect.
- an event is perceived in the same way by all parallel components: events are broadcast.
- events can be tested for presence, waited upon, or used to preempt a reactive statement.
- one cannot decide that an event is absent during the current instant before the end of the instant (this is the only moment one is sure that the event has not been generated). Thus, reaction in response to event absence is always postponed to the next instant. This is the basic principle of the reactive approach.

Event configurations are boolean expressions of events: a configuration is either a simple events, either the negation *not* of a configuration, either the *and* or the *or* of two configurations. A configuration is said to be *fixed* when its value can be evaluated safely.

2.2 Concrete Syntax

Reactive Instructions

Junior reactive instructions are defined in the following way:

- `Nothing`: does nothing and terminates immediately;
- `Stop`: stops execution for the current instant and terminates at next instant;
- `If(exp,t1,t2)`: this is the standard boolean test; `exp` is a Java boolean expression atomically evaluated.
- `Seq(t1,…,tn)`: reactive instructions are put in sequence; `ti` starts as soon as `ti-1` terminates;
- `Par(t1,…,tn)`: reactive instructions are put in parallel; the instruction terminates when all `ti` are terminated; Par is a synchronous non-deterministic operator: at each instant, all branches are executed in an arbitrary order;
- `Loop(t)`: this is an infinite loop: `t` is restarted as soon as it terminates;

- `Repeat(n,t)`: this is a finite loop where constant `n` is the number of times `t` is run up to termination;
- `RepeatExt(exp,t)`: this is a finite loop where the value of `exp` is the number of times `t` is run up to termination;
- `Generate(S)`: generates event S and terminates immediately;
- `Await(c)`: stops while configuration `c` is not satisfied and terminates immediately when it is; the simplest configuration is an event S; it is satisfied when S is generated;
- `When(c,t,u)`: `t` is immediately executed if `c` is satisfied during the current instant, and `u` is executed at the next instant otherwise;
- `Until(c,t,u)`: preemption operator; the body `t` is executed at each instant and, after the reaction, `c` is tested; if `c` is satisfied, execution of `t` is abandoned for next instants and control goes to `u`;
- `Control(S,t)`: control by an event; `t` is executed only during instants where event S is present; at other instants, the control does not reach it;
- `EventDecl(S,t)`: declaration of a local event S the scope of which is `t`.

Execution Contexts

In Junior, reactive instructions are executed by execution contexts with syntax:

- `ExecContext(t)`: the execution machine that broadcast events and the execution of which defines instants for program t.

2.3 Abstract Syntax

For formal semantics, one considers abstract syntax instead of concrete one.

Reactive Instructions

- For the following instructions, abstract syntax coincides with concrete one:

- Nothing
- Stop
- Atom(a)
- If(exp,t1,t2)
- Loop(t)
- Repeat(n,t)
- RepeatExt(exp,t)
- Generate(S)
- Await(c)

- When(c,t,u)
- Control(S,t)

• Abstract syntax only considers a binary operator for sequence:

- Seq(t,u)

• In the same way, abstract syntax considers a binary parallel operator; moreover, the operator holds the termination flags (defined later in section 3.1) of its two branches (α is the termination flag of t and β is the one of u):

- $\text{Par}_{\alpha,\beta}$(t,u)

• Abstract syntax adds a new form to the preemption Until operator. In this new form, written Until$*$, the body has already reacted but decision of actual preemption is still pendant:

- Until(c,t,u)
- Until$*$(c,t,u)

• An auxiliary information is added to local event declarations, to store local event values:

- EventDecl-(S,t)
- EventDecl+(S,t)

In these terms, sign + indicates that the local event is generated, and sign - that it is not.

Instants

One defines a new top-level instruction to cyclically run an instruction t while suspended, and to detect the end of the current instant:

- Instant(t)

Execution Contexts

The abstract syntax for execution contexts is:

- ExecContext(t)

Translation from Concrete to Abstract Syntax

The translation from concrete to abstract syntax is described as follows:

- `Nothing` -> Nothing
- `Stop` -> Stop

- `Atom(a)`	->	Atom(a)
- `If(exp,t1,t2)`	->	If(exp,t1,t2)
- `Seq(t1,…,tn)`	->	Seq(t_1,Seq(…,Seq(t_{n-1},t_n)…)
- `Par(t1,…,tn)`	->	$Par_{SUSP,SUSP}$(t_1,$Par_{SUSP,SUSP}$(…,$Par_{SUSP,SUSP}$(t_{n-1},t_n)…)
- `Loop(t)`	->	Loop(t)
- `Repeat(n,t)`	->	Repeat(n,t)
- `RepeatExt(exp,t)`	->	RepeatExt(exp,t)
- `Generate(S)`	->	Generate(S)
- `Await(c)`	->	Await(c)
- `When(c,t,u)`	->	When(c,t,u)
- `Until(c,t,u)`	->	Until(c,t,u)
- `Control(S,t)`	->	Control(S,t)
- `EventDecl(S,t)`	->	EventDecl-(S,t)
- `ExecContext(t)`	->	ExecContext(t)

The sequence and parallel n-ary operators of the concrete syntax are composed out of the binary ones in the abstract syntax. Moreover, in the abstract parallel operator, the two termination flags are initially set to SUSP (see below). Finally, local event in local declarations are initially not generated (sign -).

3. REWRITE Semantics

The basic semantics of Junior has the standard format of conditional rewriting rules[Pl].

3.1 Format of Rewritings

One writes:

$$t,E \xrightarrow{\alpha} t',E'$$

to means that instruction t executed in environment E transforms (one also says *rewrites*) in t', with E becoming E', and α returned as termination flag. There are 3 possible termination flags:

- TERM means that execution is terminated for the current instant and that nothing remains to do for the next instant;
- STOP means that execution is terminated for the current instant but that something remains to do at next instant;
- SUSP means that execution is not terminated for the current instant and thus must be resumed during it.

Environments

An environment E has the following components:

1. a set containing present events;

2. a boolean flag **eoi**(E) which is true if the end of the current instant has been decided, and false otherwise.

3. a boolean flag **move**(E) which is set to true to indicate that some change has appeared in the system; in this case, the end of the current instant must be dalayed to let the system possibility to react to this change.

The following notations are defined:

- To test if an event S is present in E, one simply writes $S \in E$;
- E+S is the environment equal to E except that event S is added to it; E-S is equal to E except that S is removed from it;
- E/F[S] is equal to E+S if $S \in F$, and is equal to E-S otherwise; E/F[S] is thus equal to E, except for S which is determined by F;
- $\gamma(\alpha,\beta)$ equals SUSP if either α or β is equal to SUSP, equals TERM if both α and β are equals to TERM, and equals STOP otherwise; it is defined by the array:

β \ α	SUSP	STOP	TERM
SUSP	SUSP	SUSP	SUSP
STOP	SUSP	STOP	STOP
TERM	SUSP	STOP	TERM

- $\delta 1(\alpha,\beta)$ equals SUSP if α is STOP and β is STOP or TERM, and equals α otherwise;
- $\delta 2(\alpha,\beta)$ equals SUSP if β is STOP and α is STOP or TERM, and equals β otherwise.
- E[**move** = v] is the environment equals to E except that the **move** flag is set to v.

3.2 Basic Statements

Nothing

Nothing immediately terminates and does nothing. The rule is:

$$\text{Nothing},E \xrightarrow{\text{TERM}} \text{Nothing},E$$

Stop

The Stop statement stops execution for the current instant, and nothing remains to be done at next instant:

$$\text{Stop},E \xrightarrow{\text{STOP}} \text{Nothing},E$$

Atoms

Atoms terminate immediately and runs a Java statement given as parameter:

$$\text{Atom(a)},E \xrightarrow{\text{TERM}} \text{Nothing},E$$

Sequence

The sequence is defined by two rules, depending on the termination of the first branch.

- If the first branch terminates, then the second one is immediately run:

$$\frac{t,E \xrightarrow{\text{TERM}} t',E' \qquad u,E' \xrightarrow{\alpha} u',E''}{Seq(t,u),E \xrightarrow{\alpha} u',E''}$$

- If the first branch is not terminated, then so is the sequence:

$$\frac{t,E \xrightarrow{\alpha} t',E' \qquad \alpha \neq \text{TERM}}{Seq(t,u),E \xrightarrow{\alpha} Seq(t',u),E'}$$

Parallelism

Parallelism is synchronous: the two branches run during each instant; it is non-deterministic: the execution order is not specified.

- If both branches are suspended (which is the initial situation at each instant), then they are both executed *in an arbitrary order*. The first rule corresponds to execute the first branch, then the second; it is called **Merge**:

$$\frac{t,E \xrightarrow{\alpha} t',E' \quad u,E' \xrightarrow{\beta} u',E''}{Par_{SUSP,SUSP}(t,u),E \xrightarrow{\gamma(\alpha,\beta)} Par_{\delta 1(\alpha,\beta),\delta 2(\alpha,\beta)}(t',u'),E''}$$

The second rule corresponds to execute the second branch, then the first one; it is called **InvMerge**:

$$\frac{u,E \xrightarrow{\beta} u',E' \quad t,E' \xrightarrow{\alpha} t',E''}{Par_{SUSP,SUSP}(t,u),E \xrightarrow{\gamma(\alpha,\beta)} Par_{\delta 1(\alpha,\beta),\delta 2(\alpha,\beta)}(t',u'),E''}$$

- If there is only one suspended branch, then it is run:

$$\frac{\beta \neq SUSP \quad t,E \xrightarrow{\alpha} t',E'}{Par_{SUSP,\beta}(t,u),\ E \xrightarrow{\gamma(\alpha,\beta)} Par_{\delta 1(\alpha,\beta),\delta 2(\alpha,\beta)}(t',u),E'}$$

$$\frac{\alpha \neq SUSP \quad u,E \xrightarrow{\beta} u',E'}{Par_{\alpha,SUSP}(t,u),E \xrightarrow{\gamma(\alpha,\beta)} Par_{\delta 1(\alpha,\beta),\delta 2(\alpha,\beta)}(t,u'),E'}$$

Loop

- A loop executes its body and rewrites in a sequence if it does not terminate immediately:

$$\frac{t,E \xrightarrow{\alpha} t',E' \quad \alpha \neq \text{TERM}}{\text{Loop}(t),E \xrightarrow{\alpha} \text{Seq}(t',\text{Loop}(t)),E'}$$

- When the loop body terminates immediately, the loop is restarted:

$$\frac{t,E \xrightarrow{\text{TERM}} t',E' \quad \text{Loop}(t),E' \xrightarrow{\alpha} t'',E''}{\text{Loop}(t),E \xrightarrow{\alpha} t'',E''}$$

Repeat

- Repeat terminates immediately if the counter is less or equal to zero:

$$\frac{n \leq 0}{\text{Repeat}(n,\ t),E \xrightarrow{\text{TERM}} \text{Nothing},E}$$

- Otherwise, the semantics is the one of a sequence:

$$\frac{n > 0 \quad \text{Seq}(t,\text{Repeat}(n-1,t)),E \xrightarrow{\alpha} t',E'}{\text{Repeat}(n,t),E \xrightarrow{\alpha} t',E'}$$

Extended Repeat

The semantics of an extended repeat is the one of a standard repeat after evaluation of a Java integer expression:

$$\frac{\textbf{eval}(\text{exp}) = n \quad \text{Repeat}(n,t),E \xrightarrow{\alpha} t',E'}{\text{RepeatExt}(\text{exp},t),E \xrightarrow{\alpha} t',E'}$$

Boolean Test

The boolean test is defined by two rules, depending on the evaluation of a Java boolean expression.

- the first branch is immediately run if evaluation returns true:

$$\frac{\textbf{eval}(\text{exp}) = \text{true} \quad t,E \xrightarrow{\alpha} t',E'}{\text{If}(\text{exp},t,u),E \xrightarrow{\alpha} t',E'}$$

- the second branch is immediately run if evaluation returns false:

$$\frac{\textbf{eval}(\text{exp}) = \text{false} \quad u,E \xrightarrow{\alpha} u',E'}{\text{If}(\text{exp},t,u),E \xrightarrow{\alpha} u',E'}$$

3.3 Event Statements

Configurations

A configuration has one of the following forms:

- a *positive* configuration, that is an event S;
- a *negative* configuration, that is the negation “not C” of a configuration;
- a conjunction “C1 and C2” of two configurations;
- a disjunction “C1 or C2” of two configurations.

Two functions **eval** and **fixed** are defined for configurations:

- **eval**(C,E) returns the value of configuration C in the environment E:

  - **eval**(S,E) ≡ S ∈ E
  - **eval**(not C,E) ≡ not **eval**(C,E)
  - **eval**(C1 and C2,E) ≡ **eval**(C1,E) and **eval**(C2,E)
  - **eval**(C1 or C2,E) ≡ **eval**(C1,E) or **eval**(C2,E)

- **fixed**(C,E) is true when configuration C can be evaluated in the environment E:

  - **fixed**(S,E) ≡ S ∈ E or **eoi**(E)
  - **fixed**(not C,E) ≡ **fixed**(C,E)
  - **fixed**(C1 and C2,E) ≡ (**fixed**(C1,E) and **fixed**(C2,E))
    or (**fixed**(C1,E) and not **eval**(C1,E))
    or (**fixed**(C2,E) and not **eval**(C2,E))
  - **fixed**(C1 or C2,E) ≡ (**fixed**(C1,E) and **fixed**(C2,E))
    or (**fixed**(C1,E) and **eval**(C1,E))
    or (**fixed**(C2,E) and **eval**(C2,E))

Note that in the basic case of an event S, **fixed**(S,E) is true if S is present or if the end of instant has been decided; this last case means that S is absent. Three auxiliary functions are also defined:

- **sat**(C,E) ≡ **fixed**(C,E) and **eval**(C,E)
- **unsat**(C,E) ≡ **fixed**(C,E) and not **eval**(C,E)

- **unknown**(C,E) ≡ not **fixed**(C,E)

Note that **unknown**(C,E) is true if and only if **sat**(C,E) and **unsat**(C,E) are both false, and that in the basic case of an event S, one has:

- **sat**(S,E) = true means that S is in E: S is present;
- **unsat**(S,E) = true means that S is not in E and that **eoi**(E) is true: S is absent.

Generate

A Generate statement adds the generated event in the environment and immediately terminates:

$$\text{Generate}(S),E \xrightarrow{\text{TERM}} \text{Nothing},E'$$

In this rule, E' is equal to E+S with **move**(E') set to true.

Events Tests

- The `then` branch is executed if the configuration is satisfied; execution is immediate if satisfaction occurs before the end of the current instant, and is delayed to the next instant otherwise:

$$\frac{\textbf{sat}(C,E) \text{ and } \textbf{eoi}(E) = \text{false} \quad t,E \xrightarrow{\alpha} t',E'}{\text{When}(C,t,u),E \xrightarrow{\alpha} t',E'}$$

$$\frac{\textbf{sat}(C,E) \text{ and } \textbf{eoi}(E)}{\text{When}(C,t,u),E \xrightarrow{\text{STOP}} t,E}$$

- The `else` branch is chosen if the configuration is not satisfied; execution is immediate if unsatisfaction occurs before the end of the current instant, and is delayed to the next instant otherwise:

$$\frac{\textbf{unsat}(C,E) \text{ and } \textbf{eoi}(E) = \text{false} \quad u,E \xrightarrow{\alpha} u',E'}{\text{When}(C,t,u),E \xrightarrow{\alpha} u',E'}$$

$$\frac{\textbf{unsat}(C,E) \text{ and } \textbf{eoi}(E)}{\text{When}(C,t,u),E \xrightarrow{\text{STOP}} u,E}$$

Note that the two previous rules returning STOP when **eoi**(E) is true basically forbid immediate reaction to events absences.

- The test is suspended if the configuration is unknown:

$$\frac{\textbf{unknown}(C,E)}{\text{When}(C,t,u),E \xrightarrow{\text{SUSP}} \text{When}(C,t,u),E}$$

Await

- Await terminates if the configuration is satisfied; termination is immediate if satisfaction occurs before the end of the current instant, and is delayed to the next instant otherwise :

$$\frac{\textbf{sat}(C,E) \text{ and not } \textbf{eoi}(E)}{\text{Await}(C),E \xrightarrow{\text{TERM}} \text{Nothing},E}$$

$$\frac{\textbf{sat}(C,E) \text{ and } \textbf{eoi}(E)}{\text{Await}(C),E \xrightarrow{\text{STOP}} \text{Nothing},E}$$

- Await stops if the configuration is unsatisfied:

$$\frac{\textbf{unsat}(C,E)}{\text{Await}(C),E \xrightarrow{\text{STOP}} \text{Await}(C),E}$$

- Await is suspended if the configuration is unknown:

$$\frac{\textbf{unknown}(C,E)}{\text{Await}(C),E \xrightarrow{\text{SUSP}} \text{Await}(C),E}$$

Control

- The body is executed if the controlling event is present:

$$\frac{\textbf{sat}(S,E) \quad t,E \xrightarrow{\alpha} t',E'}{\text{Control}(S,t),E \xrightarrow{\alpha} \text{Control}(S,t'),E'}$$

- Control stops if the event is absent:

$$\frac{\textbf{unsat}(S,E)}{\text{Control}(S,t),E \xrightarrow{\text{STOP}} \text{Control}(S,t),E}$$

- Control is suspended if the event is unknown:

$$\frac{\textbf{unknown}(S,E)}{\text{Control}(S,t),E \xrightarrow{\text{SUSP}} \text{Control}(S,t),E}$$

Until

- Until behaves as the body if it does not stop:

$$\frac{t,E \xrightarrow{\alpha} t',E' \quad \alpha \neq \text{STOP}}{\text{Until}(C,t,u),\ E \xrightarrow{\alpha} \text{Until}(C,t',u),E'}$$

- If the body stops, Until behaves as the auxiliary instruction Until$*$ (considered below):

$$\frac{t,E \xrightarrow{\text{STOP}} t',E' \quad \text{Until}*(C,t',u),E' \xrightarrow{\alpha} v,E''}{\text{Until}(C,t,u),E \xrightarrow{\alpha} v,E''}$$

Note that the body t is executed in both rules; preemption of Until is said to be *weak*, by contrast with the *strong* preemption used in the synchronous approach, which basically implies instantaneous reaction to absence.

Until*

The rules for Until* are the following:

- The handler is immediately executed if the configuration is satisfied before the end of instant:

$$\frac{\textbf{sat}(C,E) \quad \text{not } \textbf{eoi}(E) \quad u,E \xrightarrow{\alpha} u',E'}{\text{Until}*(C,t,u),E \xrightarrow{\alpha} u',E'}$$

- Until* stops and rewrites in the handler, if the configuration is satisfied while end of instant is true:

$$\frac{\textbf{sat}(C,E) \quad \textbf{eoi}(E)}{\text{Until}*(C,t,u),E \xrightarrow{\text{STOP}} u,E}$$

- Until* stops and rewrites in Until, if the configuration is unsatisfied:

$$\frac{\textbf{unsat}(C,E)}{\text{Until}*(C,t,u),E \xrightarrow{\text{STOP}} \text{Until}(C,t,u),E}$$

- Until* is suspended while the configuration is unknown:

$$\frac{\textbf{unknown}(C,E)}{\text{Until}*(C,t,u),E \xrightarrow{\text{SUSP}} \text{Until}*(C,t,u),E}$$

EventDecl

The local event is not generated in EventDecl- and present in EventDecl+. The local event is set to the appropriate value before body execution and it is saved after. The value of the event is always left unchanged in the external environment.

- If the body suspends, then the value of the local event value is stored in the produced term:

$$\frac{t,E-S \xrightarrow{\text{SUSP}} t',E' \quad S \notin E'}{\text{EventDecl}-(S,t),E \xrightarrow{\text{SUSP}} \text{EventDecl}-(S,t'),E'/E[S]}$$

$$\frac{t,E-S \xrightarrow{\text{SUSP}} t',E' \quad S \in E'}{\text{EventDecl}-(S,t),E \xrightarrow{\text{SUSP}} \text{EventDecl}+(S,t'),E'/E[S]}$$

$$\frac{t,E+S \xrightarrow{\text{SUSP}} t',E'}{\text{EventDecl}+(S,t),E \xrightarrow{\text{SUSP}} \text{EventDecl}+(S,t'),E'/E[S]}$$

- When the body terminates or stops, then the local event is reset for the next instant:

$$\frac{t,E - S \xrightarrow{\alpha} t',E' \quad \alpha = \text{TERM or } \alpha = \text{STOP}}{\text{EventDecl} - (S,t),E \xrightarrow{\alpha} \text{EventDecl} - (S,t'),E'/E[S]}$$

$$\frac{t,E + S \xrightarrow{\alpha} t',E' \quad \alpha = \text{TERM or } \alpha = \text{STOP}}{\text{EventDecl} + (S,t),E \xrightarrow{\alpha} \text{EventDecl} - (S,t'),E'/E[S]}$$

3.4. Execution Context

Execution context rewritings have the form: **e** $\Rightarrow^{b}$ e′ meaning that reaction of the execution context e leads to the new execution context e'; b is a boolean which is true if the execution context is completely terminated.

Execution Context

An execution context executes one instant of its program in a new fresh environment.

$$\frac{\text{Instant}(t),\text{Fresh} \xrightarrow{\alpha} t',E}{\text{ExecContext}(t) \Rightarrow^{b} \text{ExecContext}(t')}$$

In this rule:

- Fresh is the environment with an empty event set and such that **eoi**(Fresh) and **move**(Fresh) are both false.
- b is true if α is TERM; it is false otherwise.

Instant

Execution of an instruction during one instant means cyclic execution while it is suspended. Moreover, when execution suspends, end of instant is detected if no new event was generated during the execution.

- The instant is terminated when the instruction is stopped or terminated:

$$\frac{t,E \xrightarrow{\text{TERM}} t',E'}{\text{Instant}(t),E \xrightarrow{\text{TERM}} \text{Nothing},E'}$$

$$\frac{t,E \xrightarrow{\text{STOP}} t',E'}{\text{Instant}(t),E \xrightarrow{\text{STOP}} t',E'}$$

- Execution is immediately restarted if SUSP is returned and end of instant is set if no new event was generated:

$$\frac{t,E \xrightarrow{\text{SUSP}} t',E' \quad \textbf{move}(E') = \text{false} \quad \text{Instant}(t'),E'[\textbf{eoi} = \text{true}] \xrightarrow{\alpha} t'',E''}{\text{Instant}(t),E \xrightarrow{\alpha} t'',E''}$$

- Execution is immediately restarted if SUSP is returned and **move** is reset if a move appeared:

$$\frac{t,E \xrightarrow{SUSP} t',E' \quad \textbf{move}(E') = \text{true} \quad Instant(t'),E'[\textbf{move} = \text{false}] \xrightarrow{\alpha} t'',E''}{Instant(t),E \xrightarrow{\alpha} t'',E''}$$

4. REWRITE Implementation

We give most of the code of the implementation of the basic semantics of Junior. This direct implementation of the semantics is called REWRITE.

4.1 Rewriting Rules

Environments

An environment defines two boolean flags `eoi` and `move`, and an event set `eventSet`. Several methods are defined, which make a copy of the environment:

- `copyAdd` adds an event to the copy;
- `copyAddAndMove` adds an event to the copy and set the `move` flag to true;
- `copyRemove` removes an event from the copy;
- `copyForce` sets the value of an event according to a boolean in the copy;
- `copySetFlags` sets the `move` and `eoi` flags in the copy.

Code for class `Environment` is :

```
public class Environment
{
  final public boolean eoi, move;
  final public Vector eventSet;

  public Environment(Vector v, boolean eoi, boolean move){
       eventSet = v; this.eoi = eoi; this.move = move;
  }
  public Environment(){ this(new Vector(),false,false); }

  protected Environment add(String event,boolean b){
     Vector v = (Vector)eventSet.clone();
     if (v.contains(event) == false) v.addElement(event);
     return new Environment(v,eoi,b);
  }

  public Environment copyAdd(String event){ return add(event,move); }
  public Environment copyAddAndMove(String event){ return add(event,true); }
  public Environment copyRemove(String event){
     Vector v = (Vector)eventSet.clone();
     v.removeElement(event);
     return new Environment(v,eoi,move);
  }
  public Environment copyForce(boolean b,String event){
        return b ? copyAdd(event) : copyRemove(event) ;
  }
  public Environment copySetFlags(boolean eoi,boolean move){
        return new Environment((Vector)eventSet.clone(),eoi,move);
  }
}
```

Termination Flags

Termination flags are defined in interface `Flags`.

```
public interface Flags{ byte STOP = 0, TERM = 1, SUSP = 2; }
```

Micro States

One calls *micro-state* a triple of the form (α,t,E) where α is a termination flag, t is a term and E is an environment. All these fields are final and thus cannot be changed. Class `Micro-State` is simply defined by :

```
public class MicroState
{
  final public byte flag;
  final public Instruction term;
  final public Environment env;
  public MicroState(byte b, Instruction t, Environment e){
      flag = b; term = t;  env = e;
  }
}
```

4.2 Instructions

The only public method defined for instructions is the `rewrite` method. It is an abstract method which thus must be defined in derived classes.

Rewriting $t,E \xrightarrow{\alpha} t',E'$ means that the call t.rewrite(E) returns a micro-state s with s.flag = α, s.term = t', and s.env = E'.

The protected method `unknown` corresponds to the rewriting $t,E \xrightarrow{SUSP} t,E$.

```
public abstract class Instruction implements Flags
{
     public abstract MicroState rewrite(Environment env);

     final protected MicroState unknown(Environment env){
               return new MicroState(SUSP,this,env);
     }
}
```

Unary instructions have a body which is an instruction.

```
public abstract class UnaryInstruction extends Instruction
{
  final protected Instruction body;
  public UnaryInstruction(Instruction i){ body = i; }
}
```

Binary instructions have two instructions, called `left` and `right`.

```
public abstract class BinaryInstruction extends Instruction
{
  final protected Instruction left,right;
  public BinaryInstruction(Instruction i1,Instruction i2){
     left = i1; right = i2;
  }
}
```

An important point is that `body` in `UnaryInstruction`, and `left` and `right` in `BinaryInstruction` are final which means that they cannot be changed after construction; thus, *the structure of terms is constant and never changes at execution*.

Stop

Implementation of Stop is straightforward:

```
public class Stop extends Instruction
{
  public MicroState rewrite(Environment env){
     return new MicroState(STOP,new Nothing(),env);
  }
}
```

Sequence

In the two rules of sequence, the left instruction is first run, then the flag returned is considered; if it is TERM, then the right instruction is run, otherwise a micro-state with a new sequence is returned.

```
public class Seq extends BinaryInstruction
{
  public Seq(Instruction l,Instruction r){ super(l, r); }

  public MicroState rewrite(Environment env){
     MicroState s = left.rewrite(env);
     if (TERM == s.flag) return right.rewrite(s.env);
     return new MicroState(s.flag,new Seq(s.term,right),s.env);
  }
}
```

Parallel Operator

In the parallel operator, nondeterminism comes from the choice of the rule to apply when both flags are SUSP. It is directly coded by a method `choice` using the function `random` of Java.

The rules for `Par` are naturally coded as a sequence of tests considering the various possibilities for the two branches flags. The `result` method basically computes in one call the three functions γ, $\delta 1$, and $\delta 2$ of the semantics rules.

```
public class Par extends BinaryInstruction
{
  final protected byte leftFlag, rightFlag;
  protected boolean choice(){ return 0==(int)(Math.random()*1000)%2; }

  protected Instruction newTerm(Instruction l,Instruction r,byte lf,byte rf){
     return new Par(l,r,lf,rf);
  }
  public Par(Instruction l,Instruction r, byte lf, byte rf){
     super(l,r); leftFlag = lf; rightFlag = rf;
  }
  public Par(Instruction l,Instruction r){ this(l,r,SUSP,SUSP); }

  protected MicroState result(Instruction l,Instruction r,
                              byte lf,byte rf,Environment env){
     byte b = SUSP, nlf = lf, nrf = rf;
     if(lf!=SUSP && rf!=SUSP){
        b = (lf==TERM && rf==TERM) ? TERM : STOP;
        if (lf==STOP) nlf = SUSP;
```

```
            if (rf==STOP) nrf = SUSP;
        }
         return new MicroState(b,newTerm(l,r,nlf,nrf),env);
    }

    public MicroState rewrite(Environment env){
        if (leftFlag == SUSP && rightFlag != SUSP){
            MicroState s = left.rewrite(env);
             return result(s.term,right,s.flag,rightFlag,s.env);
        }
        if (leftFlag != SUSP && rightFlag == SUSP){
            MicroState s = right.rewrite(env);
             return result(left,s.term,leftFlag,s.flag,s.env);
        }
        // Both branches are suspended
        MicroState ls, rs;
        Environment newEnv;
        if (choice()){
            ls = left.rewrite(env);
            rs = right.rewrite(ls.env);
            newEnv = rs.env;
        }else{
            rs = right.rewrite(env);
            ls = left.rewrite(rs.env);
            newEnv = ls.env;
        }
         return result(ls.term,rs.term,ls.flag,rs.flag,newEnv);
    }
}
```

Note that nondeterminism is explicitly introduced by the `choice` method (the algorithm for random boolean generation can certainly be improved). Note also that the two flags `leftFlag` and `rightFlag` are final: the state of an instruction never changes.

Loops

A loop rewrites as a new sequence if its body does not return TERM, and otherwise it re-runs the body.

```
public class Loop extends UnaryInstruction
{
    public Loop(Instruction i){ super(i); }

    public MicroState rewrite(Environment env){
        MicroState s = body.rewrite(env);
        if (TERM == s.flag) return new Loop(body).rewrite(s.env);
        return new MicroState(s.flag,new Seq(s.term,new Loop(body)),s.env);
    }
}
```

Configurations

Configurations are defined with the four methods `fixed`, `eval`, `sat`, and `unsat`. The function called *unknown* in the semantics of configurations is not implemented as it simply corresponds to the case where both `sat` and `unsat` return false. The class `Config` of configurations is abstract; we do not give code for derived classes as it is straightforward from the definitions above.

```
abstract public class Config
{
    abstract public boolean fixed(Environment env);
    abstract public boolean eval(Environment env);
    final public boolean sat(Environment env){ return fixed(env) && eval(env); }
```

```
  final public boolean unsat(Environment env){
      return fixed(env) && !eval(env);
  }
}
```

Await

The rewrite rule of Await tests for the configuration satisfaction; if it is satisfied, it tests for the end of the current instant and terminates or stops accordingly; in both cases, it rewrites in Nothing. If the configuration is not satisfied, it stops and rewrites in itself. Finally, SUSP is returned when the configuration is unknown.

```
public class Await extends Instruction
{
  final protected Config config;
  public Await(Config c){ config = c; }
  public Await(String s){ this(new PosConfig(s)); }

  public MicroState rewrite(Environment env){
      if (config.sat(env)){
            return new MicroState((env.eoi ? STOP : TERM),new Nothing(),env);
      }
       if (config.unsat(env)){ return new MicroState(STOP,this,env); }
       // neither sat nor unsat means unknown
       return unknown(env);
  }
}
```

Local Events

Local event declarations are split into two classes corresponding to the two instructions `EventDeclNeg` and `EventDeclPos`. In both cases, the external event with same name is saved before executing the body of the declaration, and it is restored after execution.

When the body is finished, the declaration rewrites in `EventDeclNeg`, to be ready for the next instant with a local non-generated event.

We only give code for `EventDeclNeg` (code for `EventDeclPos` is very similar).

```
public class EventDeclNeg extends UnaryInstruction
{
  final protected String event;

  public EventDeclNeg(String s, Instruction t){ super(t); event = s; }

  public MicroState rewrite(Environment env){
      boolean present = env.eventSet.contains(event);
      MicroState s = body.rewrite(env.copyRemove(event));
      Instruction res;
      if (SUSP == s.flag && s.env.eventSet.contains(event))
           res = new EventDeclPos(event,s.term);
      else res = new EventDeclNeg(event,s.term);
       return new MicroState(s.flag,res,s.env.copyForce(present,event));
  }
}
```

4.3 Execution Contexts

Instants

The `Instant` instruction re-executes its program while it is suspended, and it detects the end of the current instant (setting then `eoi` to true) when there is no new generated event during one execution step, that is when `move` is still false at the end of the step.

```
public class Instant extends UnaryInstruction
{
    public Instant(Instruction t){ super(t); }

    public MicroState rewrite(Environment env){
       MicroState s = body.rewrite(env);
       if (STOP == s.flag) return s;
       if (TERM == s.flag) return new MicroState(TERM,new Nothing(),s.env);
       boolean eoi = (false == s.env.move);
       return new Instant(s.term).rewrite(s.env.copySetFlags(eoi,false));
    }
}
```

Execution Contexts

`ExecContext` performs **e** $\Rightarrow$ e′ rewritings of a program embedded in an `Instant` instruction. This rewriting is implemented by a method named `react`.

```
public class ExecContext implements Flags
{
    protected Instruction body;
    public ExecContext(Instruction t){ body = t; }
    public boolean react(){
       MicroState s = new Instant(body).rewrite(new Environment());
       body = s.term;
       return (TERM == s.flag);
    }
}
```

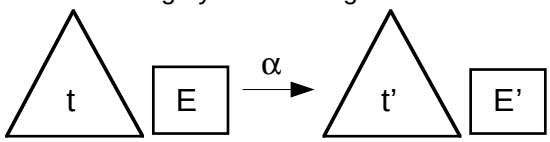
4.4 Conclusion

The code for the complete Junior REWRITE semantics is about 400 lines long. It exactly mimics the semantics rules. There is a strong invariant in the implementation: instructions never change during execution, because all their state variables are final.

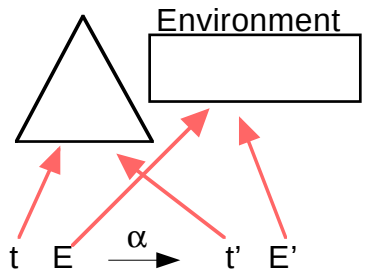
Closeness to the semantics is an advantage as one gets a reference implementation. But this is also a drawback as it performs no optimisation at all. For example, new instructions are created at each rewriting step. Moreover, each global rewriting step starts from the top and explores the whole program. The purpose of the TURBO implementation we are going to define in section 6 is to avoid these drawbacks while preserving most of the benefits of semantics formalisation. It is based on the REPLACE semantics that we introduce now.

5. REPLACE Semantics

In a rewriting $t,E \xrightarrow{\alpha} t',E'$ of the REWRITE semantics, t' and E' are distinct from t and E. One could represent the rewriting by the drawing:



The direct implementation of the semantics creates a lot of new terms and new environments during evaluation. We introduce a variant of the semantics, called *REPLACE* , which purpose is to limit creation of new items. More precisely, in the REPLACE semantics a term always rewrite in a term of the same form and the environment is shared by the two terms; thus, to rewrite a term means to change its content, not its structure. Basically, one uses references instead of instances of instructions and environments; rewritings have the form (references are represented by grey arrows):



We now give most of the rules of the REPLACE semantics, and briefly describe the implementation.

5.1 REPLACE Rules

The rules for Nothing, Control, and Par are left unchanged as these instructions always rewrites in themselves.

Stop

A boolean is added to the abstract syntax of Stop to keep trace of its termination (true means that the instruction is terminated, false that it is not) :

$$\text{Stop(false)},E \xrightarrow{\text{STOP}} \text{Stop(true)},E$$

$$\text{Stop(true)},E \xrightarrow{\text{TERM}} \text{Stop(true)},E$$

Atoms

In the same way, a boolean is introduced in Atoms to keep trace of the action execution (it

is false if the action is to be performed, true if it is already done).

- In the following rule action a is performed:

$$\mathrm{Atom(false,a),E} \xrightarrow{\mathrm{TERM}} \mathrm{Atom(true,a),E}$$

- Action a is not performed in:

$$\mathrm{Atom(true,a),E} \xrightarrow{\mathrm{TERM}} \mathrm{Atom(true,a),E}$$

Sequence

The abstract syntax for sequence is changed to keep trace of the left branch termination; false means that it is not yet terminated, true means that it is.

- If the left branch is not terminated, then so is the sequence:

$$\frac{\mathrm{t,E} \xrightarrow{\alpha} \mathrm{t',E'} \quad \alpha \neq \mathrm{TERM}}{\mathrm{Seq(false,t,u),E} \xrightarrow{\alpha} \mathrm{Seq(false,t',u),E'}}$$

- If the left branch terminates, then the right one is immediately run:

$$\frac{\mathrm{t,E} \xrightarrow{\mathrm{TERM}} \mathrm{t',E'} \quad \mathrm{u,E'} \xrightarrow{\alpha} \mathrm{u',E''}}{\mathrm{Seq(false,t,u),E} \xrightarrow{\alpha} \mathrm{Seq(true,t',u'),E''}}$$

- If left branch is already terminated, then the right branch is run:

$$\frac{\mathrm{u,E} \xrightarrow{\alpha} \mathrm{u',E'}}{\mathrm{Seq(true,t,u),E} \xrightarrow{\alpha} \mathrm{Seq(true,t,u'),E'}}$$

Loop

One deeply changes the way loops semantics is defined. Actually, we introduce a reset function on terms, used to reset loop bodies when terminated. The reset function has the effect to recursively set to false all booleans introduced in the abstract syntax to store instruction states. For example, reset(Stop(b)) = Stop(false) and reset(Seq(b,t1,t2)) = Seq(false,reset(t1), reset(t2)).

The rules for loop are now:

- A loop simply executes its body if it does not terminate immediately:

$$\frac{\mathrm{t,E} \xrightarrow{\alpha} \mathrm{t',E'} \quad \alpha \neq \mathrm{TERM}}{\mathrm{Loop(t),E} \xrightarrow{\alpha} \mathrm{Loop(t'),E'}}$$

- When the loop body terminates, the body is reset before re-execution:

$$\frac{\mathrm{t,E} \xrightarrow{\mathrm{TERM}} \mathrm{t',E'} \quad \mathrm{Loop(reset(t')),E'} \xrightarrow{\alpha} \mathrm{t'',E''}}{\mathrm{Loop(t),E} \xrightarrow{\alpha} \mathrm{t'',E''}}$$

Event Tests

One adds to the abstract syntax of When:

- a first boolean which is false if the configuration is not yet evaluated;
- a second boolean which stores the configuration value.

- The following rules are used when the configuration is not already evaluated:

$$\frac{\textbf{sat}(C,E) \text{ and } \textbf{eoi}(E) = \text{false} \quad t,E \xrightarrow{\alpha} t',E'}{\text{When}(\text{false},b,C,t,u),E \xrightarrow{\alpha} \text{When}(\text{true},\text{true},C,t',u),E'}$$

$$\frac{\textbf{sat}(C,E) \text{ and } \textbf{eoi}(E)}{\text{When}(\text{false},b,C,t,u),E \xrightarrow{\text{STOP}} \text{When}(\text{true},\text{true},C,t,u),E}$$

$$\frac{\textbf{unsat}(C,E) \text{ and } \textbf{eoi}(E) = \text{false} \quad u,E \xrightarrow{\alpha} u',E'}{\text{When}(\text{false},b,C,t,u),E \xrightarrow{\alpha} \text{When}(\text{true},\text{false},C,t,u'),E'}$$

$$\frac{\textbf{unsat}(C,E) \text{ and } \textbf{eoi}(E)}{\text{When}(\text{false},b,C,t,u),E \xrightarrow{\text{STOP}} \text{When}(\text{true},\text{false},C,t,u),E}$$

$$\frac{\textbf{unknown}(C,E)}{\text{When}(\text{false},b,C,t,u),E \xrightarrow{\text{SUSP}} \text{When}(\text{false},b,C,t,u),E}$$

- The corresponding branch is run if the configuration is already evaluated:

$$\frac{t,E \xrightarrow{\alpha} t',E'}{\text{When}(\text{true},\text{true},C,t,u),E \xrightarrow{\alpha} \text{When}(\text{true},\text{true},C,t',u),E'}$$

$$\frac{u,E \xrightarrow{\alpha} u',E'}{\text{When}(\text{true},\text{false},C,t,u),E \xrightarrow{\alpha} \text{When}(\text{true},\text{false},C,t,u'),E'}$$

Until

One adds to the abstract syntax of Until:

- a first boolean which is true if the handler has to be run.
- a second boolean which is true if the body has stopped (this corresponds to Until*).

- The following rule is used when body execution does not stop:

$$\frac{t,E \xrightarrow{\alpha} t',E' \quad \alpha \neq \text{STOP}}{\text{Until}(\text{false},\text{false},C,t,u),\ E \xrightarrow{\alpha} \text{Until}(\text{false},\text{false},C,t',u),E'}$$

- When body execution stops, semantics is given by the rules corresponding to the previous Until* term (second boolean set to true):

$$\frac{t,E \xrightarrow{STOP} t',E' \quad Until(false,true,C,t',u),E' \xrightarrow{\alpha} v,E''}{Until(false,false,C,t,u),E \xrightarrow{\alpha} v,E''}$$

- Until behaves as the handler if the first boolean is true:

$$\frac{u,E \xrightarrow{\alpha} u',E'}{Until(true,b,C,t,u),E \xrightarrow{\alpha} Until(true,b,C,t,u'),E'}$$

Until*

The following rules correspond to the previous Until* term:

$$\frac{\mathbf{sat}(C,E) \quad \mathbf{eoi}(E) = false \quad u,E \xrightarrow{\alpha} u',E'}{Until(false,true,C,t,u),E \xrightarrow{\alpha} Until(true,true,C,t,u'),E'}$$

$$\frac{\mathbf{sat}(C,E) \quad \mathbf{eoi}(E)}{Until(false,true,C,t,u),E \xrightarrow{STOP} Until(true,false,C,t,u),E}$$

$$\frac{\mathbf{unsat}(C,E)}{Until(false,true,C,t,u),E \xrightarrow{STOP} Until(false,false,C,t,u),E}$$

$$\frac{\mathbf{unknown}(C,E)}{Until(false,true,C,t,u),E \xrightarrow{SUSP} Until(false,true,C,t,u),E}$$

EventDecl

For local events, one simply codes the sign of the declaration with a boolean (true means positive, false negative).

- Rules corresponding to EventDecl- are:

$$\frac{t,E - S \xrightarrow{SUSP} t',E' \quad S \notin E'}{EventDecl(false,S,t),E \xrightarrow{SUSP} EventDecl(false,S,t'),E'/E[S]}$$

$$\frac{t,E - S \xrightarrow{SUSP} t',E' \quad S \in E'}{EventDecl(false,S,t),E \xrightarrow{SUSP} EventDecl(true,S,t'),E'/E[S]}$$

$$\frac{t,E - S \xrightarrow{\alpha} t',E' \quad \alpha = TERM \text{ or } \alpha = STOP}{EventDecl(false,S,t),E \xrightarrow{\alpha} EventDecl(false,S,t'),E'/E[S]}$$

- Rules corresponding to EventDecl+ are:

$$\frac{t,E + S \xrightarrow{SUSP} t',E'}{EventDecl(true,S,t),E \xrightarrow{SUSP} EventDecl(true,S,t'),E'/E[S]}$$

$$\frac{t,E+S \xrightarrow{\alpha} t',E' \quad \alpha = \text{TERM or } \alpha = \text{STOP}}{\text{EventDecl}(\text{true},S,t),E \xrightarrow{\alpha} \text{EventDecl}(\text{false},S,t'),E'/E[S]}$$

5.2 REPLACE Implementation

In the REPLACE implementation, there is only one environment which is shared by all instructions. Thus, in class `Instruction` one introduces a field of type `Environment` to store it, and one defines a method `bind` which must be called before any rewriting and which purpose is to set it.

One also defines a new method `reset` to deal with loops; a loop resets its body when it is terminated.

Micro-states of the REWRITE implementation are no more used and are simply replaced by return flags (which are bytes).

```
public abstract class Instruction implements Flags
{
     protected Environment env;
     protected void bind(Environment e){ env = e; }

     abstract public byte rewrite();
     abstract protected void reset();
}
```

The `reset` method propagates to binary instructions components (and also to the body of unary instructions).

```
public abstract class BinaryInstruction extends Instruction
{
  final protected Instruction left, right;
  public BinaryInstruction(Instruction i1,Instruction i2){
     left = i1; right = i2;
  }

  protected void bind(Environment e){
      super.bind(e); left.bind(e); right.bind(e);
  }
  protected void reset(){ left.reset(); right.reset(); }
}
```

Stop

Stop uses a boolean flag to return STOP at the first instant, and TERM at future ones.

```
public class Stop extends Instruction
{
  protected boolean terminated = false;
  protected void reset(){ terminated = false; }

  public byte rewrite(){
     if (terminated) return TERM; else terminated = true;
     return STOP;
  }
}
```

Sequence

A boolean called `leftTerminated` is set to true when the left instruction is terminated. In this case, the control immediately switches to the right branch.

```
public class Seq extends BinaryInstruction
{
  protected boolean leftTerminated = false;
  public Seq(Instruction l,Instruction r){ super(l,r); }
  protected void reset(){ super.reset(); leftTerminated = false; }

  public byte rewrite(){
      if (leftTerminated) return right.rewrite();
      byte s = left.rewrite();
      if (s == TERM){ leftTerminated = true; return right.rewrite(); }
      return s;
  }
}
```

Loops

The body is reset when terminated, to be immediately rerun.

```
public class Loop extends UnaryInstruction
{
  public Loop(Instruction i){ super(i); }

  public byte rewrite(){
      byte s = body.rewrite();
      if (s != TERM) return s;
      body.reset();
      return rewrite();
  }
}
```

Await

A boolean flag keeps trace of the termination of an Await statement. This is necessary because Await stops when the configuration is satisfied at the end of the current instant; in this case, one records that the instruction must terminate at next instant.

```
public class Await extends Instruction
{
  final protected Config config;
  protected boolean terminated = false;

  public Await(Config c){ config = c; }
  public Await(String s){ this(new PosConfig(s)); }
  protected void reset(){ terminated = false; }

  public byte rewrite(){
      if (terminated) return TERM;
      if (config.sat(env)){
         terminated = true;
         return env.eoi() ? STOP : TERM;
      }
      if (config.unsat(env)) return STOP;
      return SUSP;
  }
}
```

Execution Contexts

At construction, an execution context binds its program to a new environment; then, at

each reaction, it resets the environment. The code is:

```
public class ExecContext implements Flags
{
    protected Instant instant;
    public ExecContext(Instruction t){
        instant = new Instant(t); instant.bind(new Environment());
    }
    public boolean react(){
        byte res = instant.rewrite();
        instant.env.reset();
        return (TERM == res);
    }
```

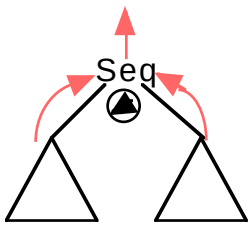
5.3 Conclusion

The REPLACE rules are very close to those of REWRITE. Actually, REPLACE can be seen as a simple remake of REWRITE, in which one avoids to duplicate terms and environments. The REPLACE semantics can thus be considered as a reference semantics for Junior, as well as REWRITE. However, in REPLACE, *rewriting basically means changing the state of terms and environments, not their structures*.

In the REPLACE implementation each rewriting step still continues to start from the top of the program and explores the whole of it. This is optimised in the TURBO implementation which we are going to describe now.

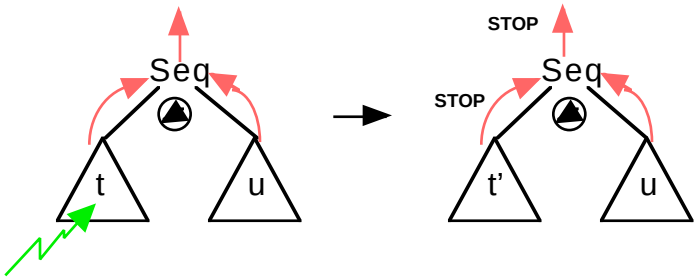
6. TURBO Implementation

The REPLACE implementation does not create new instructions. It has a second advantage: as the program structure does not change, execution can be *processed in reverse order*, starting from the bottom (the leaves of the abstract syntax tree) and returning to the top of the program. For example, consider the sequence operator Seq. In the replacement implementation, the `leftTerminated` boolean is introduced to indicate if the left branch is terminated or not. One can represent a sequence by the following drawing in which the grey arrows point to father nodes in the abstract syntax tree, and the triangle in the circle indicates which branch is to be executed first (the left one in the drawing):

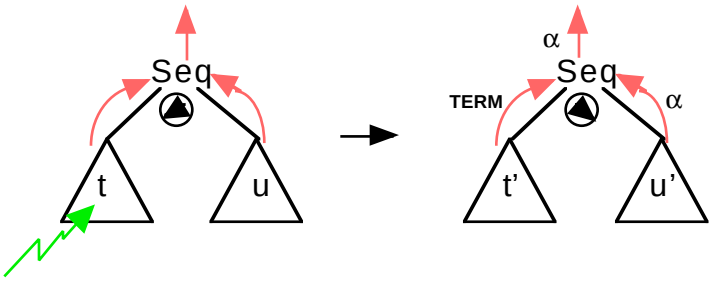


It then becomes possible to directly fire a sub-term of the sequence, the result of which is transmitted upward.

For example, the rewriting $\dfrac{t,E \xrightarrow{STOP} t',E'}{Seq(t,u),E \xrightarrow{STOP} Seq(t',u),E'}$ is represented by:



And the rewriting $\dfrac{t,E \xrightarrow{TERM} t',E' \quad u,E' \xrightarrow{\alpha} u',E''}{Seq(t,u),E \xrightarrow{\alpha} u',E''}$ corresponds to:

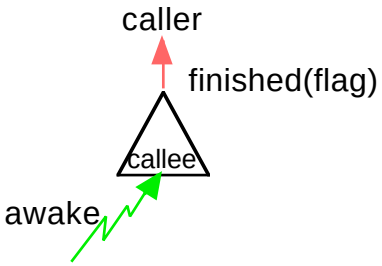


Note the change of the triangle position in the right tree of the drawing, reflecting the setting of `leftTerminated` to true.

Using this approach, the whole program has not to be explored at each rewriting step: execution can only consider sub-terms waiting for generated events because only these sub-terms are actually run.

Upward Changes

Activation of sub-term is named `awake` and retransmission to the caller is named `finished`:



Local Events Processing

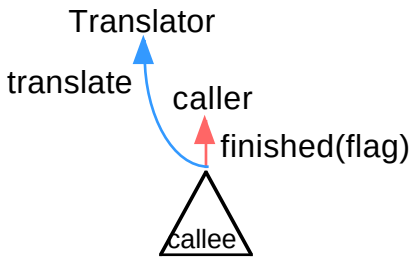
Upward replacements cause troubles with local events declarations, as the save/restore mechanism used to implement them in the semantics is basically a descending strategy, not an ascending one. We solve the problem by giving local events new fresh hidden names, and by introducing a translator mechanism to translate actual events names to hidden names. The notion of a `Translator` is introduced as an interface that only defines the `translate` method:

```
public interface Translator { public String translate(String s); }
```

A reference to a translator is given by calling the `bind` method of instructions which prototype thus becomes:

```
void bind(Environment env,Instruction c,Translator t);
```

One thus represents instructions by the following drawing:



6.1 Implementation

We only give the main points of the TURBO implementation.

Instructions

The `finished` method signals that a sub-term, named `callee`, has been rewritten, and it returns the flag obtained, when different from SUSP.

To awake an instruction (`awake` method) means to rewrite it and to signal the caller with the result if the instruction is not suspended.

Finally, the `unknown` method has a parameter which is the configuration on which the instruction is blocked. It adds itself to the list of instructions blocked on the configuration which is managed by the environment.

```
public abstract class Instruction implements Flags
{
     protected Instruction caller;
     protected Translator trans;
     protected Environment env;
     protected void bind(Environment e,Instruction i,Translator t){
        env = e; caller = i; trans = t;
     }
     // awake
```

```
    public void awake(){
       byte s = rewrite();
       if (s != SUSP) caller.finished(this,s);
    }
    // basically, retransmit
    public void finished(Instruction callee,byte s){
       caller.finished(callee,s);
    }
    // rewriting
    abstract public byte rewrite();
    // register as unknown
    protected byte unknown(Config config){
       config.fallAsleep(this,env);
       return SUSP;
    }
    // reset
    protected void reset(){}
}
```

Sequence

The `rewrite` method of the sequence is the same as in the REPLACE implementation. The `finished` method of the sequence, when called by the left instruction, runs the right instruction, if the left one was terminated. The resulting flag is transmitted to the caller, if different from SUSP.

```
public void finished(Instruction callee,byte s){
   if (callee==left){
     leftTerminated = (s == TERM);
     if (leftTerminated){
        byte r = right.rewrite();
        if (r != SUSP) caller.finished(this,r);
     }else caller.finished(this,STOP);
   }
   else if (callee==right) caller.finished(this,s);
}
```

Instant

Implementation of the Instant instruction deeply departs from REPLACE. It is based on a three-phases algorithm:

1. the program tree is rewritten from top to bottom in the first phase which exactly corresponds to the REPLACE semantics;
2. instructions are awaken in the second phase when events on which they are blocked are generated;
3. in the third phase, all non generated events are set absent, and remaining blocked instructions are awaken.

```
public class Instant extends UnaryInstruction implements Translator
{
   protected byte flag;
   public Instant(Instruction t){ super(t); }

   public String translate(String s){ return s; }
   public void finished(Instruction callee, byte s){ flag = s; }

   public byte rewrite(){
      // first step : standard semantics
      flag = body.rewrite();
      if (flag != SUSP) return flag;
      // second step : upward activations
```

```
        env.awakeEveryBody();
        if (flag != SUSP) return flag;
        // third step : events absence processing
        env.eoi = true;
        env.notifyAllEvents();
        env.awakeEveryBody();
        return flag;
    }
}
```

EventDecl

`EventDecl` implements `Translator` by defining the `translate` method (`hidden` is a new fresh name):

```
public String translate(String s){
    return event.equals(s) ? hidden : trans.translate(s);
}
```

An `EvenDecl` instruction is the translator of its body:

```
protected void bind(Environment e,Instruction i,Translator t){
    super.bind(e,i,t); body.trans = this; // I'm a translator !
}
```

Then, rewriting simply means to rewrite the body :

```
public byte rewrite(){
    byte s = body.rewrite();
    if (TERM == s) env.remove(hidden);
    return s;
}
```

6.2 Conclusion

The TURBO implementation is about twice longer than the basic one (about 800 lines). It is based on the REPLACE semantics rules and introduces the way to directly fire subterms. This minimalises the number of abstract syntax tree traversals while computing reactions. Moreover, a distinct implementation of local events is introduced, based on the use of new hidden fresh names.

The TURBO implementation differs in a large extends from the basic REWRITE implementation. However, it is basically a variant of the REPLACE implementation of which it takes most of the code. Even if we are not able to formally prove equivalences of REWRITE and TURBO implementations, we have a great confidence in TURBO, as we have derived it from the basic REWRITE implementation, via the REPLACE implementation, by a small number of little transformations steps.

7. Implementation Efficiency

One compares the three implementations. We use a small example which runs the following instruction t defined by:

```
Instruction par = new Generate("s0");
for(int i = 0; i<num; i++){
    Instruction branch =
        new Seq(new Await("s"+i),
         new Seq(new Generate("s"+(i+1)),new PrintAtom("s"+i)));
    par = new Par(branch,par);
}

Instruction t = new Loop(new Seq(par,new Stop()));
```

Instruction `t` is made of `num` parallel components and has the form:

```
new Par(…
    new Par(new Seq(new Await("s2"),
              new Seq(new Generate("s3"),new PrintAtom("s2"))),
    new Par(new Seq(new Await("s1"),
              new Seq(new Generate("s2"),new PrintAtom("s1"))),
    new Par(new Seq(new Await("s0"),
              new Seq(new Generate("s1"),new PrintAtom("s0"))),
             new Generate("s0")))…)
```

In order to compare the three implementations, one uses in the three cases the same Par operator restricted to chose the left branch first (**Merge** rule only).

Best Case for TURBO

Execution with the left branch first is the worst possible case for REWRITE and REPLACE, as it implies a maximum number `num` of program traversals. It is the best case for TURBO, which is able to fire the `num` branches one after the other. Results are given in milliseconds by the following array:

	REWRITE	REPLACE	TURBO
num = 100	366	163	160
num = 200	1135	401	394
num = 1000	143442	5599	3043

Best Case for REPLACE

We change the construction, replacing `res = new Par(t,res);` by `res = new Par(res,t);` Then, we obtain:

	REPLACE	TURBO
num = 100	136	139
num = 200	285	300
num = 1000	2048	2802

Absence of Events

By suppressing the generation of s0, one gets:

	REPLACE	TURBO
num = 100	15	30
num = 200	16	32
num = 1000	24	65

In conclusion: TURBO is clearly much better when a lot of events are generated during instants. REPLACE and TURBO are very close in other situations.

7. The Dj Model

The Dj model (for *Distributed Junior* model) is the distributed version of Junior. It basically corresponds to the introduction of asynchrony in Junior. Non-atomic actions, called *extern actions*, are introduced with the syntax:

- Extern(a)

We only give the changes with the Junior REWRITE semantics.

Instructions

There is a new possible flag in rewriting rules; it is called DELAY and corresponds to an asynchronous action:

- DELAY means that execution must be resumed during the current instant which thus cannot be finished immediately.

Asynchronism

The **Async** rule states that any instruction t can be delayed at any step:

$$t,E \xrightarrow{\text{DELAY}} t,E$$

Extern Actions

- As atoms, extern actions can terminate immediately. The rule **AtomicExtAct** is:

$$\text{Extern(a)},E \xrightarrow{\text{TERM}} \text{Nothing},E$$

- Extern actions can also delay execution before completion. The rule **PartialExtAct** is:

$$\text{Extern(a)},E \xrightarrow{\text{DELAY}} \text{Extern(b)},E$$

In this rule, the residual b of a must be executed during next micro-steps. Action a can thus be intermixed with others extern actions.

Parallelism

The only change for Par is to deal with DELAY, which has a higher priority than SUSP. Actually, the change concerns the functions γ, $\delta 1$, and $\delta 2$ which become:

- γ is defined by the array:

β \ α	DELAY	SUSP	STOP	TERM
DELAY	DELAY	DELAY	DELAY	DELAY
SUSP	DELAY	SUSP	SUSP	SUSP
STOP	DELAY	SUSP	STOP	STOP
TERM	DELAY	SUSP	STOP	TERM

- $\delta 1(\alpha,\beta)$ equals DELAY if α is STOP and β is STOP or TERM, and equals α otherwise;
- $\delta 2(\alpha,\beta)$ equals DELAY if β is STOP and α is STOP or TERM, and equals β otherwise.

Instant

A new rule is introduced for Instant in order to deal with DELAY. It behaves as if SUSP where returned, except that end of instant detection is inhibited.

- Execution is immediately restarted if it returns DELAY:

$$\frac{t,E \xrightarrow{\text{DELAY}} t',E' \quad \text{Instant}(t'),E' \xrightarrow{\alpha} t'',E''}{\text{Instant}(t),E \xrightarrow{\alpha} t'',E''}$$

Note that end of instant is never set in this rule, even if no new event has appeared during rewriting of t.

8. Relations with SugarCubes

Junior is actually a descendant of SugarCubes[BS] which is a proposal to mix the reactive approach with Java. SugarCubes implementation used the same approach as REPLACE. In order to use SugarCubes in the telecom context, we have felt the need for an efficient implementation of it and we have developed the TURBO implementation. However, TURBO quickly appears **not** to be an implementation of SugarCubes (see next section), but the one of a more compact formalism which is actually Junior.

8.1 SugarCubes

We just consider three aspects of SugarCubes relevant to the comparison with Junior: the deterministic Merge parallel operator of SugarCubes, the processing of loops, and the difficulty of sub-terms firing with SugarCubes.

Merge

SugarCubes parallelism is restricted to a deterministic version (`Merge`) of `Par`. This corresponds to forbid the **MergeInv** rule of Junior. All others Junior reactive instructions are similar in SugarCubes (`ExecContext` basically corresponds to SugarCubes reactive machines). Thus, Junior can be seen as a variant of SugarCubes, with a non-deterministic parallel operator.

Implementation of the `Merge` operator of SugarCubes is straightforward from the basic REWRITE semantics of Junior. Actually, it is sufficient to redefine the `choice` method in a way that it always returns true.

```
public class Merge extends Par
{
     public Merge(Instruction l, Instruction r, byte lf, byte rf){
        super(l,r,lf,rf);
     }

     protected boolean choice(){ return true; }

     protected Instruction newTerm(Instruction l,Instruction r,byte lf,byte rf){
        return new Merge(l,r,lf,rf);
     }
}
```

Loops

A loop which during one single instant cyclically runs its never stooping body is a cause of trouble as it prevent the rewriting process to converge. Such a loops is called an *instantaneous loops*. SugarCubes detects instantaneous loops and force their bodies to stop; this is not the case in Junior in which the user is free to implement its own strategy for detecting and possibly processing them.

Firing sub-terms

SugarCubes is implemented using a replacement technique. However, the firing of sub-terms is not possible for SugarCubes as it is for Junior. Consider the following definition for instruction t:

```
Instruction t1 = new Seq(new Await(new PosConfig("e")),new PrintAtom("a1"));
Instruction t2 = new Seq(new Await(new PosConfig("f")),new Generate("e"));
Instruction t3 = new Seq(new Await(new PosConfig("e")),new PrintAtom("a2"));
Instruction t4 = new Generate("f");
Instruction t = new Merge(t1,new Merge(t2,new Merge(t3,t4)));
```

With the semantics of SugarCubes, the only possible output is a2a1. First, t1, then t2, then t3 suspend; then, event f is generated. As a new event has been generated, end of instant is not decided and t is rerun; t1 suspends another time, t2 generate e, as f is present, and t3 prints a2, as e is present. End of instant is again postponed, as e is generated, and t is rerun, which prints a1.

With the firing sub-terms approach, execution is as follows: first, configuration of t1, t2, and t3 are posted; then, f is generated, which awakes t2; execution of t2 generates e which awakes both t1 and t3. At that point, there is no guarantee that t2 is run before t1 and, thus, a difference becomes possible with the semantics of SugarCubes. The point is that it seems to be very difficult to maintain the good ordering of execution of awaken instructions. The problem of preserving such an ordering disappears in Junior because it is basically nondeterministic.

8.2 SugarCubes based Formalisms

Amongst the formalisms designed on top of SugarCubes and implemented with it are Re-

active Scripts and Distributed Reactive Machines.

Reactive Scripts[BH] gives the flexibility and dynamicity of interpretation to the reactive approach in Java. It introduces a small language in which one can program scripts reacting to broadcast events. Reactive Scripts has been used to implement Icobj Programming[Bo] which is a graphical framework introducing the notion of behavioural icons (called *icobjs*) and giving ways to build new behaviours by combining them.

Reactive Distributed Machines[SBH] (RDM) defines reactive machines distributed over the network but sharing the same instants and communicating with broadcast events. RDM is implemented in Java using the RMI mechanism.

Before ending this section, we compare Junior, Dj, SugarCubes, Reactive Scripts, and RDM, considering the following instruction A:

```
Par(
  Seq(Await("e"),Atom(a1)),
  Seq(Await("e"),Atom(a2)),
  Seq(Generate("e"),Atom(a3))
)
```

Junior

With the semantics of Junior, one can prove that a3 is always executed first and that there are only 2 possible sequences of actions: a3 a1 a2 or a3 a2 a1.

Dj

With the semantics of Dj, one can prove that all possible interleavings of a1, a2, and a3 are possible; actually, there are 6 possible outputs: a1 a2 a3, a1 a3 a2, a2 a1 a3, a2 a3 a1, a3 a1 a2, or a3 a2 a1.

SugarCubes

The SugarCubes instruction corresponding to A is:

```
Merge(
  Seq(Await("e"),Atom(a1)),
Merge(
  Seq(Await("e"),Atom(a2)),
  Seq(Generate("e"),Atom(a3))
)
```

Now, there is only one possible output: a3 a1 a2.

Reactive Scripts

The reactive script equivalent to A is:

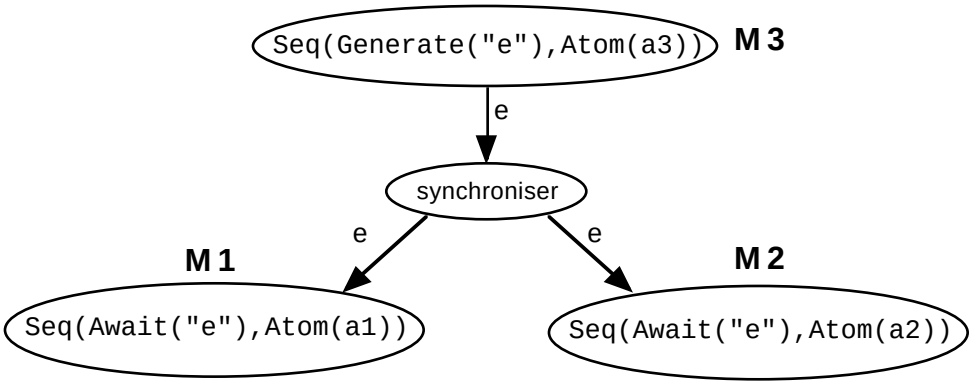
```
  await e; {a1}
||
  await e; {a2}
||
  generate e; {a3}
```

It basically follows Junior semantics: a3 a1 a2 and a3 a2 a1 are the only possible outputs.

Distributed Reactive Machines

Consider 3 distributed reactive machines M1, M2, and M3, with broadcast event e such that:

- M1 executes the program `Seq(Await("e"),Atom(a1))`
- M2 executes `Seq(Await("e"),Atom(a2))`,
- M3 executes `Seq(Generate("e"),Atom(a3))`:

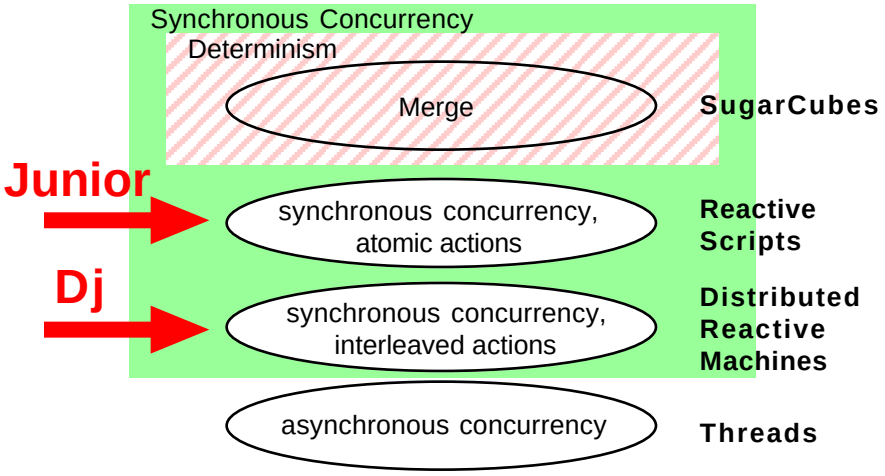


Suppose that actions a1, a2, and a3 are executed by the same object, in order to visualise their execution order; then, the possible outputs are the 6 ones of the Dj semantics.

Suppose now, that Atom instructions are replaced by Extern instructions of Dj in the 3 machines; then the result becomes unpredictable as the codes of a1, a2, and a3 can be arbitrarily intermixed.

Conclusion

The previous example shows that Junior and Reactive Scripts on one hand, and Dj and DRM on the other hand, are basically at the same level. The following figure sums up the situation:



9. Conclusion

Junior is a kernel model for reactive programming. It basically defines concurrent reactive instructions communicating with broadcast events. At the basis of Junior is the rejection of immediate reaction to absence, which is one of the major difference with synchronous formalisms.

The `Par` parallel operator in Junior is basically non-deterministic. This is the main difference with SugarCubes which has a deterministic `Merge` parallel operator.

Junior has an efficient implementation in which the whole program has not to be walked through at each micro-step. In this implementation called TURBO, instructions stuck on events are stored in queues, and generation of an event directly fires all instructions stuck on it.

The Junior model can be extended to match the Distributed Reactive Machines Model. The new formalism, called Dj, is built from Junior in a very straightforward way.

References

[Bo] F. Boussinot, *Icobj Programming*, research report INRIA RR-3028, 1996.

[BH] F. Boussinot, L. Hazard, *Reactive Scripts*, 3rd International Conference on Real-Time Computing Systems an Applications RTCSA'96, Seoul, IEEE, pp. 270-277, 1996. Disponible en rapport de recherche INRIA RR-2868, 1996.

[BS] F. Boussinot, J-F. Susini, *The SugarCubes Tool Box - A reactive Java framework*, Software Practice & Experience, **28**(14), pp. 1531-1550, 1998.

[Pl] G. Plotkin, *A Structural Approach to Operational Semantics*, Report DAIMI FN-19, Aarhus University, 1981.

[SBH] J-F. Susini, F. Boussinot, L. Hazard, *Distributed Reactive Machines*, 5th International Conférence on Real-Time Computing Systems an Applications RTCSA'98, Hiroshima, IEEE, pp. 267-274, october 1998.

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - B.P. 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Lorraine : Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - B.P. 101 - 54602 Villers lès Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot St Martin (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, B.P. 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399